САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе № 4

«ISA. Ассемблер, дизассемблер»

Выполнил(а): Шеметов Алексей Игоревич

Номер ИСУ: 338978

студ. гр. М3134

Санкт-Петербург

2021

Что такое RISC-V?

Для начала разберемся, что такое risc?

RISC — архитектура процессора, в которой быстродействие увеличивается за счёт упрощения инструкций, чтобы их декодирование было более простым, а время выполнения — меньшим. В системах команд первых RISC-процессоров даже отсутствовали команды умножения и деления. Это также облегчает повышение тактовой частоты и делает более эффективной суперскалярность.

Характерные особенности RISC-процессоров

Фиксированная длина машинных инструкций (например, 32 бита) и простой формат команды.

Специализированные команды для операций с памятью — чтения или записи. Операции вида Read-Modify-Write («прочитать-изменить-записать») отсутствуют. Любые операции «изменить» выполняются только над содержимым регистров (т. н. архитектура load-and-store).

Большое количество регистров общего назначения (32 и более).

Отсутствие поддержки операций вида «изменить» над укороченными типами данных — байт, 16-разрядное слово. Так, например, система команд DEC Alpha содержала только операции над 64-разрядными словами, и требовала разработки и последующего вызова процедур для выполнения операций над байтами, 16- и 32-разрядными словами.

Отсутствие микропрограмм внутри самого процессора. То, что в CISC-процессоре исполняется микропрограммами, в RISC-процессоре исполняется как обыкновенный (хотя и помещённый в специальное хранилище) машинный код, не отличающийся принципиально от кода ядра ОС и приложений. Так, например, обработка отказов страниц в DEC Alpha и интерпретация таблиц страниц содержалась в так называемом PAL Code (Privileged Architecture

Library), помещённом в ПЗУ. Заменой PALCode можно было превратить процессор Alpha из 64-разрядного в 32-разрядный, а также изменить порядок байтов в слове и формат входов таблиц страниц виртуальной памяти.

RISC-V - система команд и процессорная архитектура на основе концепции RISC для микропроцессоров и микроконтроллеров. Имеет встроенные возможности для расширения списка команд и подходит для широкого круга применений.

Описание RISC-V включает сравнительно небольшое число стандартных инструкций, около 50 штук, многие из которых были типичны ещё для ранних RISC-I 1980 года. Стандартные расширения (M, A, F и D) расширяют набор на 53 инструкции, сжатый формат C определяет 34 команды. Используется 6 типов кодирования инструкций (форматов).

Система команд

В архитектуре RISC-V имеется обязательное для реализации небольшое подмножество команд (набор инструкций I — Integer) и несколько стандартных опциональных расширений.

В базовый набор входят инструкции условной и безусловной передачи управления/ветвления, минимальный набор арифметических/битовых операций на регистрах, операций с памятью (load/store), а также небольшое число служебных инструкций.

Операции ветвления не используют каких-либо общих флагов, как результатов ранее выполненных операций сравнения, а непосредственно сравнивают свои регистровые операнды. Базис операций сравнения минимален, а для поддержки комплементарных операций операнды просто меняются местами.

Базовое подмножество команд использует следующий набор регистров: специальный регистр x0 (zero), 31 целочисленный регистр общего назначения

(x1 — x31), регистр счётчика команд (PC, используется только косвенно), а также множество CSR (Control and Status Registers, может быть адресовано до 4096 CSR).

Для встраиваемых применений может использоваться вариант архитектуры RV32E (Embedded) с сокращённым набором регистров общего назначения (первые 16). Уменьшение количества регистров позволяет не только экономить аппаратные ресурсы, но и сократить затраты памяти и времени на сохранение/восстановление регистров при переключениях контекста.

При одинаковой кодировке инструкций в RISC-V предусмотрены реализации архитектур с 32, 64 и 128-битными регистрами общего назначения и операциями (RV32I, RV64I и RV128I соответственно).

Разрядность регистровых операций всегда соответствует размеру регистра, а одни и те же значения в регистрах могут трактоваться целыми числами как со знаком, так и без знака.

Нет операций над частями регистров, нет каких-либо выделенных «регистровых пар».

Операции не сохраняют где-либо биты переноса или переполнения, что приближено к модели операций в языке программирования Си. Также аппаратно не генерируются исключения по переполнению и даже по делению на 0. Все необходимые проверки операндов и результатов операций должны производиться программно.

Целочисленная арифметика расширенной точности (большей, чем разрядность регистра) должна явно использовать операции вычисления старших битов результата. Например, для получения старших битов произведения регистра на регистр имеются специальные инструкции.

Размер операнда может отличаться от размера регистра только в операциях с памятью. Транзакции к памяти осуществляются блоками, размер в байтах которых должен быть целой неотрицательной степенью 2, от одного байта до

размера регистра включительно. Операнд в памяти должен иметь «естественное выравнивание» (адрес кратен размеру операнда).

Архитектура использует только модель little-endian — первый байт операнда в памяти соответствует наименее значащим битам значений регистрового операнда.

Для пары инструкций сохранения/загрузки регистра операнд в памяти определяется размером регистра выбранной архитектуры, а не кодировкой инструкции (код инструкции один и тот же для RV32I, RV64I и RV128I, но размер операндов 4, 8 и 16 байт соответственно), что соответствует размеру указателя, типам языка программирования C size_t или разности указателей.

Для всех допустимых размеров операндов в памяти, меньших, чем размер регистра, имеются отдельные инструкции загрузки/сохранения младших битов регистра, в том числе для загрузки из памяти в регистр есть парные варианты инструкций, которые позволяют трактовать загружаемое значение как со знаком (старшим знаковым битом значения из памяти заполняются старшие биты регистра) или без знака (старшие биты регистра устанавливаются в 0).

Инструкции базового набора имеют длину 32 бита с выравниванием на границу 32-битного слова, но в общем формате предусмотрены инструкции различной длины (стандартно — от 16 до 192 бит с шагом в 16 бит) с выравниванием на границу 16-битного слова. Полная длина инструкции декодируется унифицированным способом из её первого 16-битного слова.

Для наиболее часто используемых инструкций стандартизовано применение их аналогов в более компактной 16-битной кодировке (C — Compressed extension).

Операции умножения, деления и вычисления остатка не входят в минимальный набор инструкций, а выделены в отдельное расширение (M — Multiply extension). Имеется ряд доводов в пользу разделения и данного набора на два отдельных (умножение и деление).

Стандартизован отдельный набор атомарных операций (A — Atomic extension).

Поскольку кодировка базового набора инструкций не зависит от разрядности архитектуры, то один и тот же код потенциально может запускаться на различных RISC-V архитектурах, определять разрядность и другие параметры текущей архитектуры, наличие расширений системы инструкций, а потом автоконфигурироваться для целевой среды выполнения.

Спецификацией RISC-V предусмотрено несколько областей в пространстве кодировок инструкций для пользовательских «X-расширений» архитектуры, которые поддерживаются на уровне ассемблера, как группы инструкций custom0 и custom1.

Регистры

RISC-V имеет 32 (или 16 для встраиваемых применений) целочисленных регистра. При реализации вещественных групп команд есть дополнительно 32 вещественных регистра.

Рассматривается вариант включения в стандарт дополнительного набора из 32 векторных регистров с вариативной длиной обрабатываемых значений, длина которых указывается в CSR vlenb.

Для операций над числами в бинарных форматах плавающей запятой используется набор дополнительных 32 регистров FPU (Floating Point Unit), которые совместно используются расширениями базового набора инструкций для трёх вариантов точности: одинарной — 32 бита (F extension), двойной — 64 бита (D — Double precision extension), а также четверной — 128 бит (Q — Quadruple precision extension).

ELF файлы

Что представляет собой файл ELF?

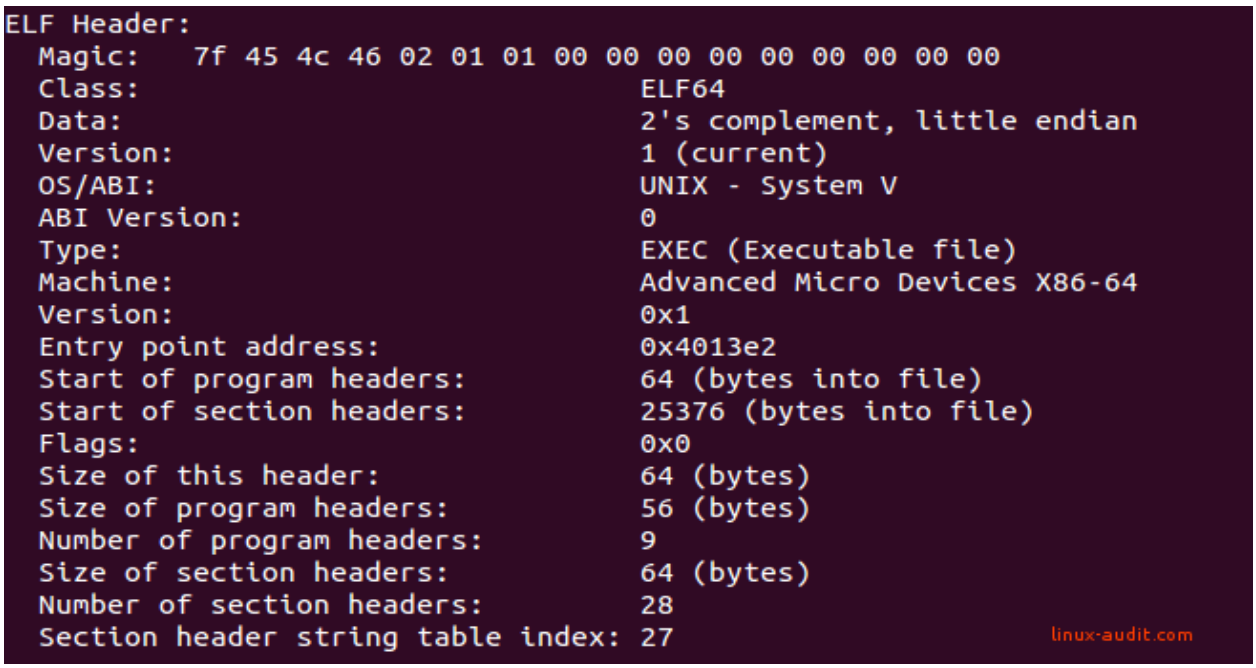
ELF — это сокращение от Executable and Linkable Format (формат исполняемых и связываемых файлов) и определяет структуру бинарных файлов, библиотек, и файлов ядра (core files). Спецификация формата позволяет операционной системе корректно интерпретировать содержащиеся в файле машинные команды. Файл ELF, как правило, является выходным файлом компилятора или линкера и имеет двоичный формат. С помощью подходящих инструментов он может быть проанализирован и изучен.

От исходника к процессу

Какую бы операционную систему мы не использовали, необходимо каким-то образом транслировать функции исходного кода на язык CPU — машинный код. Функции могут быть самыми базовыми, например, открыть файл на диске или вывести что-то на экран. Вместо того, чтобы напрямую использовать язык CPU, мы используем язык программирования, имеющий стандартные функции. Компилятор затем транслирует эти функции в объектный код. Этот объектный код затем линкуется в полную программу, путём использования линкера. Результатом является двоичный файл, который может быть выполнен на конкретной платформе и конкретном типе CPU.

Структура

В силу расширяемости ELF-файлов, структура может различаться для разных файлов. ELF-файл состоит из заголовка ELF (см. рисунок 1) и данных



«Рисунок №1 – заголовок elf файла»

Заголовок ELF

Как видно на рисунке №1, заголовок ELF начинается с «магического числа». Это «магическое число» даёт информацию о файле. Первые 4 байта определяют, что это ELF-файл (45=E,4c=L,46=F, перед ними стоит значение 7f).

Заголовок ELF является обязательным. Он нужен для того, чтобы данные корректно интерпретировались при линковке и исполнении. Для лучшего понимания внутренней работы ELF-файла, полезно знать, для чего используется эта информация.

Класс

После объявления типа ELF, следует поле класса. Это значение означает архитектуру, для которой предназначен файл. Оно может равняться 01 (32-битная архитектура) или 02 (64-битная). Здесь мы видим 02, что переводится как файл ELF64, то есть, другими словами, этот файл использует 64-битную архитектуру.

Данные

Далее идёт поле «данные», имеющее два варианта: 01 — LSB (Least Significant

Bit), также известное как little-endian, либо 02 — MSB (Most Significant Bit, big-endian). Эти значения помогают интерпретировать остальные объекты в файле. Это важно, так как разные типы процессоров по разному обрабатывают структуры данных. В примере используется LSB, так как процессор имеет архитектуру AMD64.

Версия

Затем следует ещё одно магической значение «01», представляющее собой номер версии. В настоящее время имеется только версия 01, поэтому это число не означает ничего интересного.

Полный заголовок

```
7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 |.ELF............|
02 00 3e 00 01 00 00 00 a8 2b 40 00 00 00 00 00 |..>......+@.....|
40 00 00 00 00 00 00 00 30 65 01 00 00 00 00 00 |@.......0e......|
00 00 00 00 40 00 38 00 09 00 40 00 1c 00 1b 00 |....@.8...@.....|
```

Выделенное поле определяет тип машины. Значение 3e — это десятичное 62, что соответствует AMD64.

Данные файла

Помимо заголовка, файлы ELF состоят из трёх частей.

- Программные заголовки или сегменты
- Заголовки секций или секции
- Данные

Файл ELF имеет два различных «вида». Один из них предназначен для линкера и разрешает исполнение кода (сегменты). Другой предназначен для команд и данных (секции). В зависимости от цели, используется соответствующий тип заголовка. Рассмотрим заголовок программы, который находится в исполняемых файлах ELF.

Заголовки программы

Файл ELF состоит из нуля или более сегментов, и описывает, как создать процесс, образ памяти для исполнения в рантайме. Когда ядро видит эти сегменты, оно размещает их в виртуальном адресном пространстве, используя системный вызов mmap(2). Другими словами, конвертирует заранее подготовленные инструкции в образ в памяти. Если ELF-файл является обычным бинарником, он требует эти программные заголовки, иначе он просто не будет работать. Эти заголовки используются, вместе с соответствующими структурами данных, для формирования процесса. Для разделяемых библиотек (shared libraries) процесс похож.

```
Elf file type is EXEC (Executable file)
Entry point 0x402ba8
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000015144 0x0000000000015144  R E    200000
  LOAD           0x0000000000015e00 0x0000000000615e00 0x0000000000615e00
                 0x00000000000005f8 0x00000000000214e8  RW     200000
  DYNAMIC        0x0000000000015e18 0x0000000000615e18 0x0000000000615e18
                 0x00000000000001e0 0x00000000000001e0  RW     8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x0000000000012c84 0x0000000000412c84 0x0000000000412c84
                 0x000000000000071c 0x000000000000071c  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000015e00 0x0000000000615e00 0x0000000000615e00
                 0x0000000000000200 0x0000000000000200  R      1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version
.gnu.version_r .rela.dyn .rela.plt .init .plt .text .fini .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .jcr .dynamic .got
                                                                    linux-audit.com
```

«Рисунок №2 – Программный заголовок в бинарном ELF-файле»

Заголовки секции

Заголовки секции определяют все секции файла. Как уже было сказано, эта информация используется для линковки и релокации. Секции появляются в ELF-файле после того, как компилятор GNU C преобразует код C в ассемблер, и ассемблер GNU создаёт объекты. Как показано на рисунке вверху, сегмент может иметь 0 или более секций. Для исполняемых файлов существует четыре главных секций: .text, .data, .rodata, и .bss.

.text

Содержит исполняемый код. Он будет упакован в сегмент с правами на чтение и на исполнение. Он загружается один раз, и его содержание не изменяется.

.data

Инициализированные данные, с правами на чтение и запись.

. rodata

Инициализированные данные, с правами только на чтение.

.bss

Неинициализированные данные, с правами на чтение/запись.

.symtab

Таблица символов объектного файла содержит информацию, необходимую для нахождения и перемещения символьных определений и символьных ссылок программы. Индекс таблицы символов является нижним индексом этого массива. Индекс 0 обозначает первую запись в таблице и служит неопределенным индексом символа.

Практическая часть.

В начале считаем magic numbers, чтобы проверить, что на вход подан elf файл. Далее считываем e_shoff, e_shum и e_shstrndx необходимые для нахождения начала .text и .symtab. Чтобы найти .text и .symtab распарсим header. Далее просто соотносим команды в .text и .symtab с RVC командами, переводим и записываем в заданный файл.

Листинг

Интерпретатор: python 3.8.10

main.py

```
import sys

fileOpen = True
try:
    file = open(sys.argv[1], "rb").read()
except FileNotFoundError as err:
    print("File not found")
    fileOpen = False
except IndexError as err:
    print("Usage: elf-parser.py <elf-file> <output-file>")
    fileOpen = False

def to_num(nums):
    res = 0
    t = 1
    for i in nums:
        res += int(i, 16) * t
        t *= 256
    return res

def to_name(start):
    name = ''
    j = 0
    while True:
        num = list(map(hex, file[start + j:start + j +  1]))
        if num[0][2:] == '0':
            break
        name += bytes.fromhex(num[0][2:]).decode("utf8")
        j += 1
```

```
    return name

def to_hex(num):
    res = '0x'
    for i in range(0, len(num)):
        if len(num[len(num) - 1 - i][2:]) == 1:
            res += '0'
        res += num[len(num) - 1- i][2:]
    return res

def to_bin(num):
    res = ''
    for i in reversed(num):
        res += (bin(int(i, 16))[2:].zfill(8))
    return res

def to_two(bits):
    return -int(bits[0]) << len(bits) | int(bits, 2)

def get_reg(x):
    x = int(x, 2)
    if x == 0:
        return "zero"
    elif x == 1:
        return "ra"
    elif x == 2:
        return "sp"
    elif x == 3:
        return "gp"
    elif x == 4:
        return "tp"
    elif x == 5:
        return "t0"
    elif x == 6:
        return "t1"
    elif x == 7:
        return "t2"
    elif x == 8:
        return "s0"
    elif x == 9:
        return "s1"
    elif x >= 10 and x <= 17:
        return 'a' + str(x - 10)
    elif x >= 18 and x <= 27:
```

```
        return 's' + str(x - 16)
    elif x >= 28 and x <=  31:
        return 't' + str(x - 25)
    else:
        return "ERROR"

def get_command(bytes):
    if bytes[25:] == "0110111":
        const = bytes[0:20] + '0'*12
        rd = bytes[20:25]
        return ["lui", get_reg(rd), to_two(const)]
    elif bytes[25:] == "0010111":
        const = bytes[0:20] + '0'*12
        rd = bytes[20:25]
        return ["auipc", get_reg(rd), to_two(const)]
    elif bytes[25:] == "1101111":
        const = bytes[0] + bytes[1:20] + bytes[11] + bytes[1:11] + '0'
        rd = bytes[20:25]
        return ["jal", get_reg(rd), to_two(const)]
    elif bytes[25:] == "1100111":
        if bytes[17:20] == "000":
            rd = bytes[20:25]
            rs1 = bytes[12:17]
            const = bytes[0:11]
            return ["jalr", get_reg(rd), get_reg(rs1), to_two(const)]
    elif bytes[25:] == "1100011":
        const = bytes[0] + bytes[24] + bytes[1:7] + bytes[20:24] + '0'
        rs2 = bytes[7:12]
        rs1 = bytes[12:17]
        command = []

        if bytes[17:20] == "000":
            command.append("beq")
        elif bytes[17:20] == "001":
            command.append("bne")
        elif bytes[17:20] == "100":
            command.append("blt")
        elif bytes[17:20] == "101":
            command.append("bge")
        elif bytes[17:20] == "110":
            command.append("bltu")
        elif bytes[17:20] == "111":
            command.append("bgeu")
        command.append(get_reg(rs1))
```

```
        command.append(get_reg(rs2))
        command.append(to_two(const))
        return command
    elif bytes[25:] == "0000011":
        rd = get_reg(bytes[20:25])
        rs1 = get_reg(bytes[12:17])
        const = to_two(bytes[0:12])
        command = []
        if bytes[17:20] == "000":
            command.append("lb")
        elif bytes[17:20] == "001":
            command.append("lh")
        elif bytes[17:20] == "010":
            command.append("lw")
        elif bytes[17:20] == "100":
            command.append("lbu")
        elif bytes[17:20] == "101":
            command.append("lhu")
        command.append(rd)
        command.append(rs1)
        command.append(const)
        return command
    elif bytes[25:] == "0100011":
        rs1 = get_reg(bytes[12:17])
        rs2 = get_reg(bytes[7:12])
        const = to_two(bytes[0:7] + bytes[20:25])
        command = []
        if bytes[17:20] == "000":
            command.append("sb")
        elif bytes[17:20] == "001":
            command.append("sh")
        elif bytes[17:20] == "010":
            command.append("sw")
        command.append(rs1)
        command.append(rs2)
        command.append(const)
        return command
    elif bytes[25:] == "0010011":
        rd = get_reg(bytes[20:25])
        rs1 = get_reg(bytes[12:17])
        const = to_two(bytes[0:12])
        shamt = to_two(bytes[7:12])
        command = []
```

```
        if bytes[17:20] == "001":
            return ["slli", rd, rs1, shamt]
        elif bytes[17:20] == "101":
            if bytes[0:7] == "0000000":
                return ["srli", rd, rs1, shamt]
            elif bytes[0:7] == "0100000":
                return ["srai", rd, rs1, shamt]

        if bytes[17:20] == "000":
            command.append("addi")
        elif bytes[17:20] == "010":
            command.append("slti")
        elif bytes[17:20] == "011":
            command.append("sltiu")
        elif bytes[17:20] == "100":
            command.append("xori")
        elif bytes[17:20] == "110":
            command.append("ori")
        elif bytes[17:20] == "111":
            command.append("andi")
        command.append(rd)
        command.append(rs1)
        command.append(const)
        return command
    elif bytes[25:] == "0110011":
        rd = get_reg(bytes[20:25])
        rs1 = get_reg(bytes[12:17])
        rs2 = get_reg(bytes[7:12])
        command = []

        if bytes[17:20] == "000":
            if bytes[0:7] == "0000000":
                command.append("add")
            elif bytes[0:7] == "0100000":
                command.append("sub")
        elif bytes[17:20] == "001" and bytes[0:7] == "0000000":
            command.append("sll")
        elif bytes[17:20] == "010" and bytes[0:7] == "0000000":
            command.append("slt")
        elif bytes[17:20] == "011" and bytes[0:7] == "0000000":
            command.append("sltu")
        elif bytes[17:20] == "100" and bytes[0:7] == "0000000":
            command.append("xor")
        elif bytes[17:20] == "101" and bytes[0:7] == "0000000":
```

```
            command.append("srl")
        elif bytes[17:20] == "101" and bytes[0:7] == "0100000":
            command.append("sra")
        elif bytes[17:20] == "110" and bytes[0:7] == "0000000":
            command.append("or")
        elif bytes[17:20] == "111" and bytes[0:7] == "0000000":
            command.append("and")
        command.append(rd)
        command.append(rs1)
        command.append(rs2)
        return command
    elif bytes[25:] == "1110011" and bytes[0:25] ==
"00000000000000000000000":
        return ["ecall"]
    elif bytes[25:] == "1110011" and bytes[0:25] ==
"00000000001000000000000":
        return ["ebreak"]
    else:
        return ["unknown_command"]

def get_bind(x):
    if x == 0:
        return "LOCAL"
    elif x == 1:
        return "GLOBAL"
    elif x == 2:
        return "WEAK"
    elif x == 10:
        return "LOOS"
    elif x == 12:
        return "HIOS"
    elif x == 13:
        return "LOPROC"
    elif x == 15:
        return "HIPROC"

def get_type(x):
    if x == 0:
        return "NOTYPE"
    elif x == 1:
        return "OBJECT"
    elif x == 2:
        return "FUNC"
    elif x == 3:
```

```
        return "SECTION"
    elif x == 4:
        return "FILE"
    elif x == 5:
        return "COMMON"
    elif x == 6:
        return "TLS"
    elif x == 10:
        return "LOOS"
    elif x == 12:
        return "HIOS"
    elif x == 13:
        return "LOPROC"
    elif x == 15:
        return "HIPROC"

def get_vis(x):
    if x == 0:
        return "DEFAULT"
    elif x == 1:
        return "INTERNAL"
    elif x == 2:
        return "HIDDEN"
    elif x == 3:
        return "PROTECTED"
    elif x == 4:
        return "EXPORTED"
    elif x == 5:
        return "SINGLETON"
    elif x == 6:
        return "ELIMINATE"

def get_index(x):
    if x == 0:
        return "UNDEF"
    elif x == 65280: # ff00
        return "LORESERVE"
    elif x == 65281: # ff01
        return "AFTER"
    elif x == 65311: # ff1f
        return "HIPROC"
    elif x == 65312: # ff20
        return "LOOS"
    elif x == 65343: # ff3f
```

```
        return "HIOS"
    elif x == 65521: # fff1
        return "ABS"
    elif x == 65522: # fff2
        return "COMMON"
    elif x == 65535: # ffff
        return "XINDEX"
    else:
        return x

def parsing_header(e_shoff, e_shnum, e_shstrndx):
    header_tables = to_num(e_shoff) + (to_num(e_shstrndx)) * 40
    name_table = to_num(list(map(hex, file[header_tables + 16:header_tables +
20])))
    sh_name = []

    for i in range(to_num(e_shnum)):
        sh_name.append(to_num(list(map(hex, file[to_num(e_shoff) + 40 *
i:to_num(e_shoff) + 40 * i + 4]))))

    symtab_index = -1
    text_index = -1

    index = 0
    names = []
    for i in sh_name:
        name = ''
        while True:
            num = list(map(hex, file[name_table + i:name_table + i + 1]))
            if num[0][2:] == '0':
                break
            name += bytes.fromhex(num[0][2:]).decode("utf8")
            i += 1
        names.append(name)
        if name == '.symtab':
            symtab_index = index
        if name == '.text':
            text_index = index
        index += 1

    return (
        to_num(list(map(hex, file[to_num(e_shoff) + 40 * symtab_index + 16:
                  to_num(e_shoff) + 40 * symtab_index + 20]))), symtab_index,
        to_num(list(map(hex, file[to_num(e_shoff) + 40 * text_index + 16:
```

```
                    to_num(e_shoff) + 40 * text_index + 20]))), text_index
    )

def parsing_text(text_offset, text_index, e_shoff, out):
    size = to_num(list(map(hex, file[to_num(e_shoff) + 40 * text_index +
20:to_num(e_shoff) + 40 * text_index + 24])))
    adress =  to_hex(list(map(hex, file[to_num(e_shoff) + 40 * text_index +
12:to_num(e_shoff) + 40 * text_index + 16])))

    adress = int(adress, 16)
    for i in range(size // 4):
        bytes = to_bin(list(map(hex, file[text_offset + i * 4:text_offset + i
* 4 + 4])))
        command = get_command(bytes)

        out.write("%08x "%(adress))
        if i == 0:
            out.write("%10s: "%("_start"))
        else:
            out.write("%10s  "%(" "))
        for j in range(len(command)):
            if j != 0 and j != len(command) - 1:
                out.write('%s, '%(str(command[j])))
            else:
                out.write('%s '%(str(command[j])))
        out.write('\n')

        adress += 4

def parsing_symtab(symtab_offest, symtab_index, e_shoff, e_shstrndx, out):
    idk = to_num(e_shoff) + (to_num(e_shstrndx) - 1) * 40
    start = to_num(list(map(hex, file[idk + 16:idk + 20])))

    sizeSymtab = to_num(list(map(hex, file[to_num(e_shoff) + 40 *
symtab_index + 20:to_num(e_shoff) + 40 * symtab_index + 24])))
    rowsCnt = sizeSymtab // 16

    out.write("%s %-15s %7s %-8s %-8s %-8s %6s %s\n"%("Symbol", "Value",
"Size", "Type", "Bind", "Vis", "Index", "Name"))
    for i in range(rowsCnt):
        row = []
```

```
        value = list(map(hex, file[symtab_offest + i * 16 + 4:symtab_offest +
i * 16 + 8]))
        row.append(int(to_hex(value), 16))

        size = list(map(hex, file[symtab_offest + i * 16 + 8:symtab_offest +
i * 16 + 12]))
        row.append(int(to_hex(size), 16))

        # Add Type and Bind
        tb = list(map(hex, file[symtab_offest + i * 16 + 12:symtab_offest + i
* 16 + 13]))
        tb = bin(int(tb[0], 16))[2:].zfill(8)
        # Type
        row.append(get_type(int(tb[4:8], 2)))
        # Bind
        row.append(get_bind(int(tb[0:4], 2)))

        vis = list(map(hex, file[symtab_offest + i * 16 + 13:symtab_offest +
i * 16 + 14]))
        vis = bin(int(vis[0], 16))[2:].zfill(8)
        row.append(get_vis(int(vis, 2)))

        index = list(map(hex, file[symtab_offest + i * 16 + 14:symtab_offest
+ i * 16 + 16]))
        index = to_num(index)
        row.append(get_index(index))

        name = to_num(list(map(hex, file[symtab_offest + i * 16:symtab_offest
+ i * 16 + 4])))
        if name == 0:
            row.append("")
        else:
            row.append(to_name(start + name))

        out.write("[%4i] 0x%-15X %5i %-8s %-8s %-8s %6s %s\n"%(i, row[0],
int(row[1]), row[2], row[3], row[4], row[5], row[6]))

if __name__ == "__main__" and fileOpen:
    try:
        magic_nums = list(map(hex, file[0:4]))
        if magic_nums[0] != '0x7f' or magic_nums[1] != '0x45' or
magic_nums[2] != '0x4c' or magic_nums[3] != '0x46':
            print("Unsopported file")
        else:
```

```
            try:
                out = open(sys.argv[2], 'w')

                e_shoff = list(map(hex, file[32:36]))
                e_shnum = list(map(hex, file[48:50]))
                e_shstrndx = list(map(hex, file[50:52]))

                symtab_offset, symtab_index, text_offset, text_index =
parsing_header(e_shoff, e_shnum, e_shstrndx)
                parsing_text(text_offset, text_index, e_shoff, out)
                out.write('\n')
                parsing_symtab(symtab_offset, symtab_index, e_shoff,
e_shstrndx, out)

                out.close()
            except IndexError as err:
                print("Usage: elf-parser.py <elf-file> <output-file>")
    except IndexError as err:
        print("File is empty")
```